Microkernel Construction

Inter-Process-Communication

SS2015

Tobias Stumpf

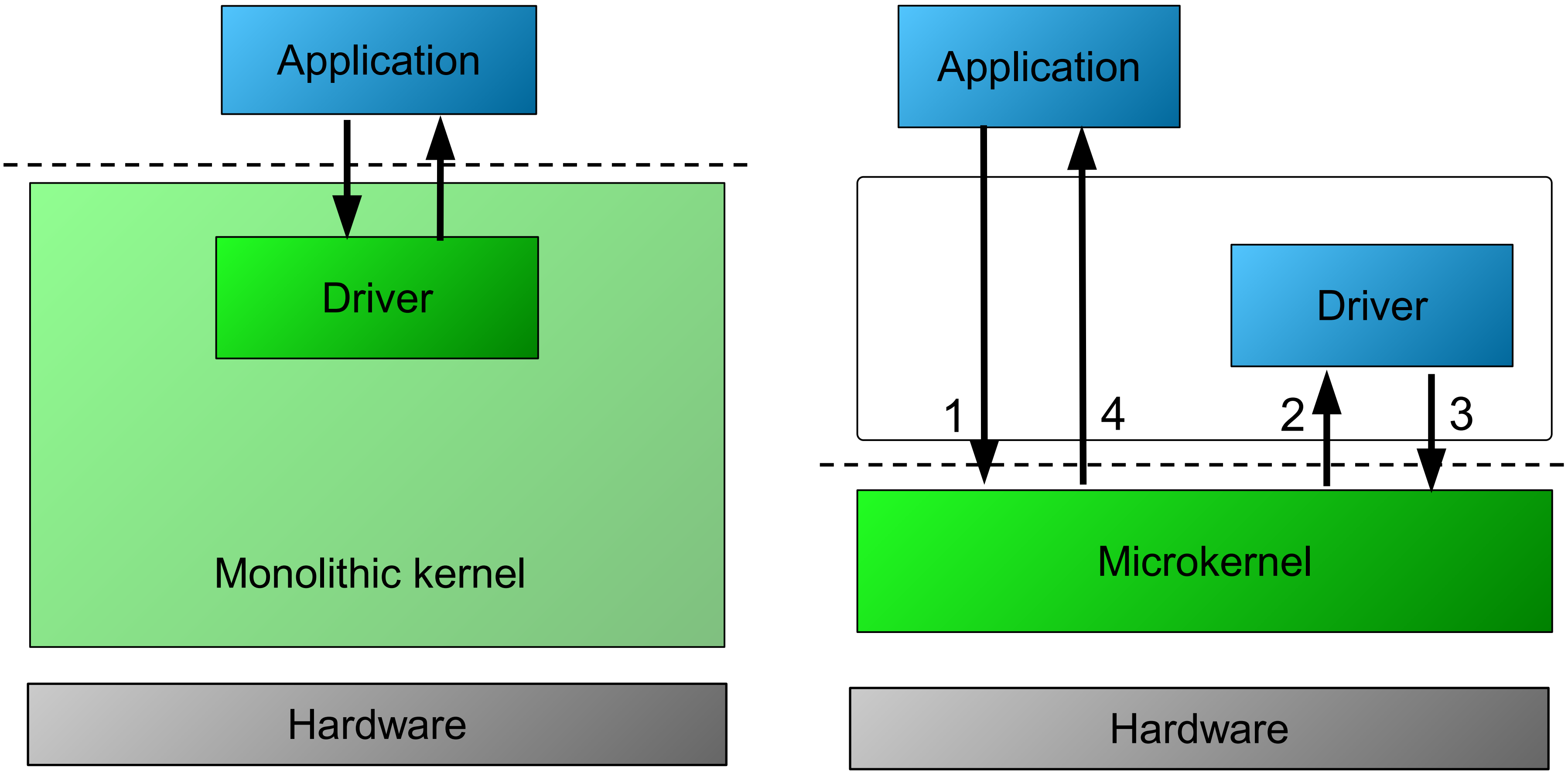
Benjamin Engel
Torsten Frenzel

TU Dresden
Operating Systems Group

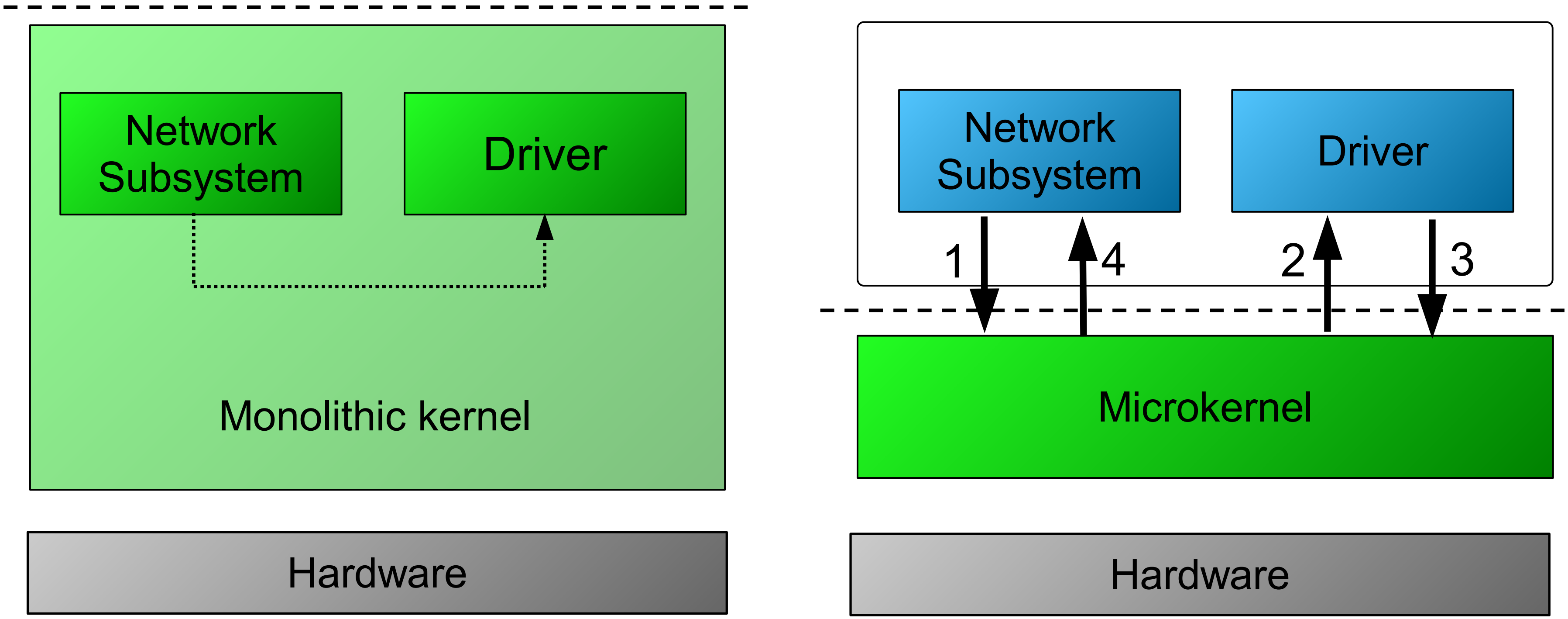
Use Cases

- Generally in microkernels:
  - Control transfer (synchronization)
  - Data transfer (send/receive data)
- Specifically
  - Grant access to resources (memory, io-ports, capabilities)
  - Manage and handle page faults, interrupts and other resources
  - Timeouts
  - Yield CPU (sleep)
- Allows feature-rich user-level protocols on top
  - Tailored for client/server communication
  - Microkernel talks some simple protocols to support user applications (page fault IPC, exception IPC)
- Optimized for performance

- System calls
  - Monolithic kernel: 2 kernel entries/exits
  - Microkernel: 4 kernel entries/exits + 2 context switches



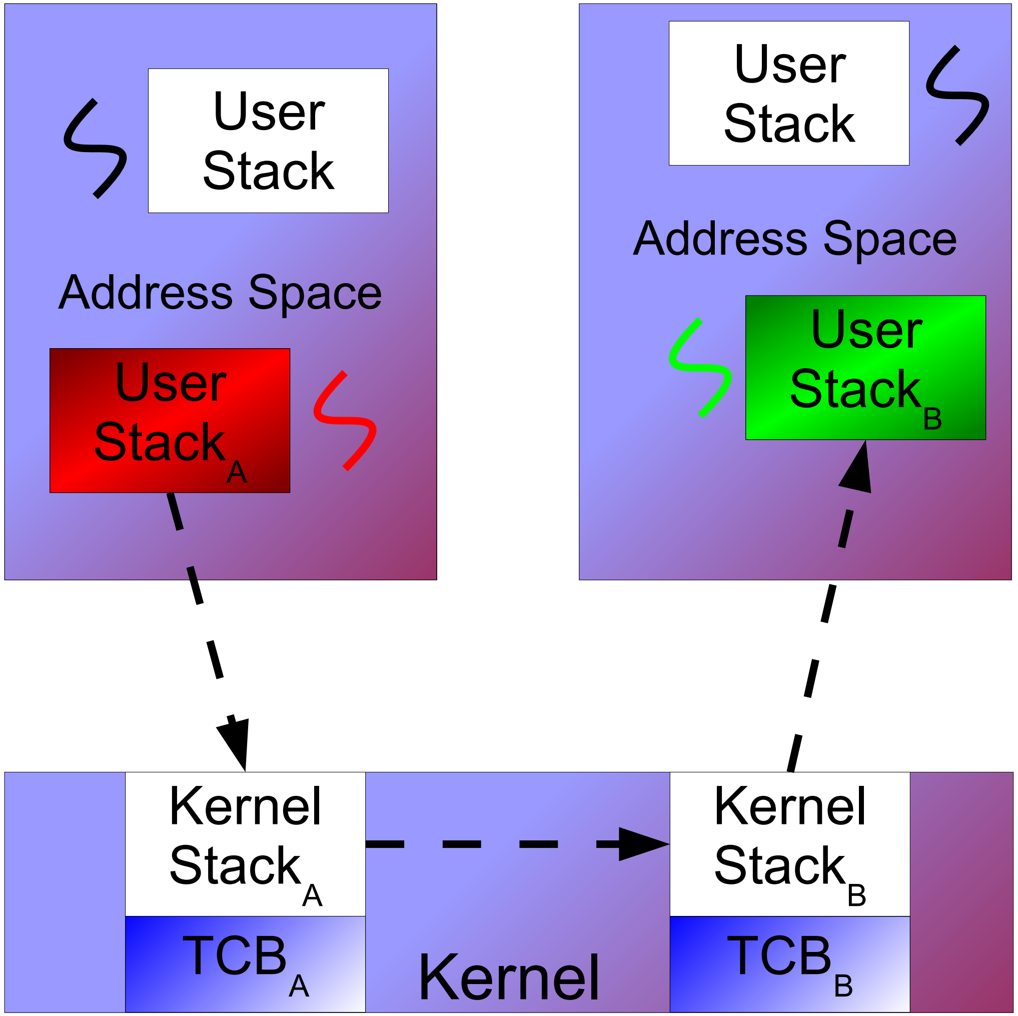
- Calls between system services
  - Monolithic kernel: 1 function call
  - Microkernel: 4 kernel entries/exits + 2 context switches



POSIX IPC Primitives

- Semaphore
  - Atomic increment and decrement of counter with wait queue
  - Used to synchronize critical sections
- Shared memory
  - Special file type, that uses physical memory
- Message queue
  - Special file type for block-based synchronous communication
- Pipe
  - Untyped, directed communication channel
- FIFO
  - Special file type similar to a pipe
  - Blocking and non-blocking mode
- Regular files
- Signals
  - Asynchronous trigger; many have predefined actions

Recap: Thread Switch

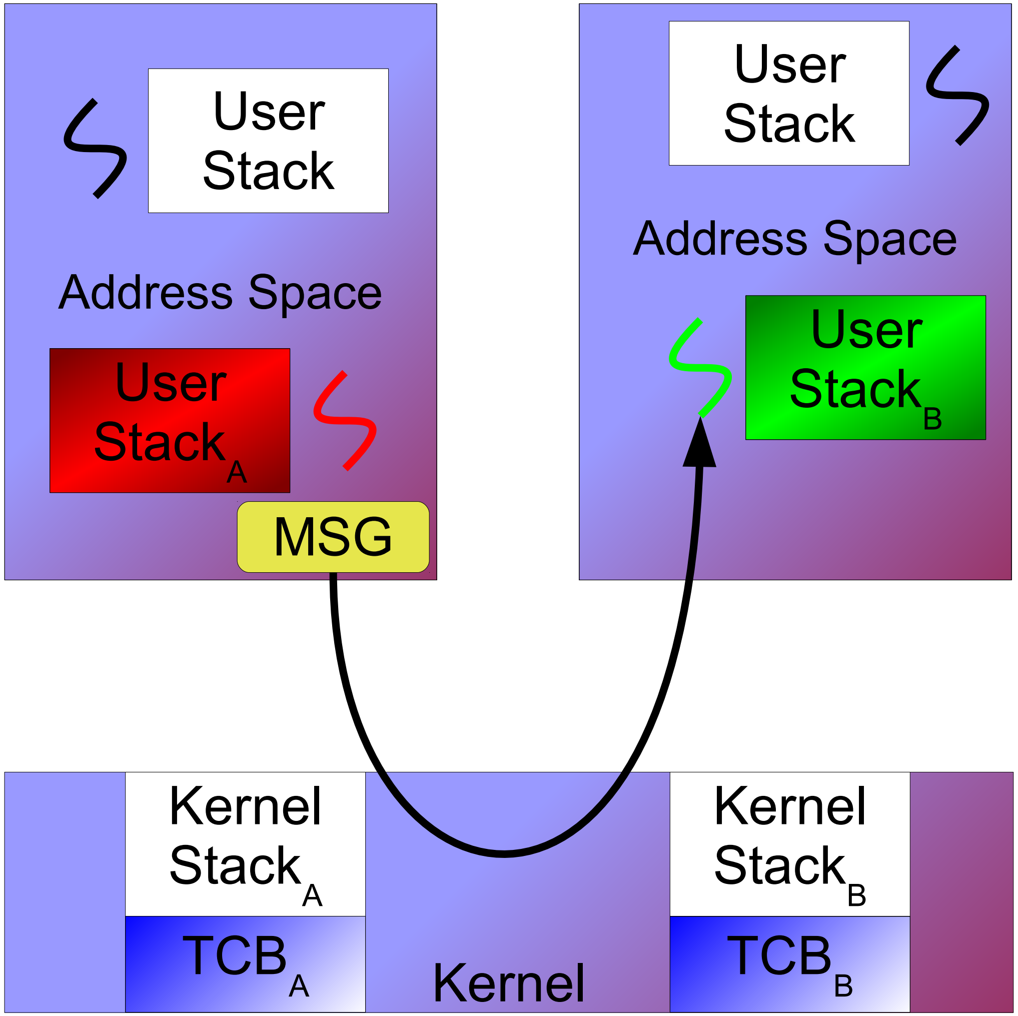


- Enter Kernel
- Switch A → B
- Exit Kernel

Benjamin Engel
Torsten Frenzel

TU Dresden Operating Systems Group

Send Message (async)



Send

- Prepare Message
- Enter Kernel
- Allocate in-kernel message buffer
- Copy message in
- Find Receiver and tag new msg available
- Exit Kernel

Receive

- Enter Kernel
- Copy message out to user memory
- Free in-kernel message buffer
- Exit Kernel

IPC Properties and Terms

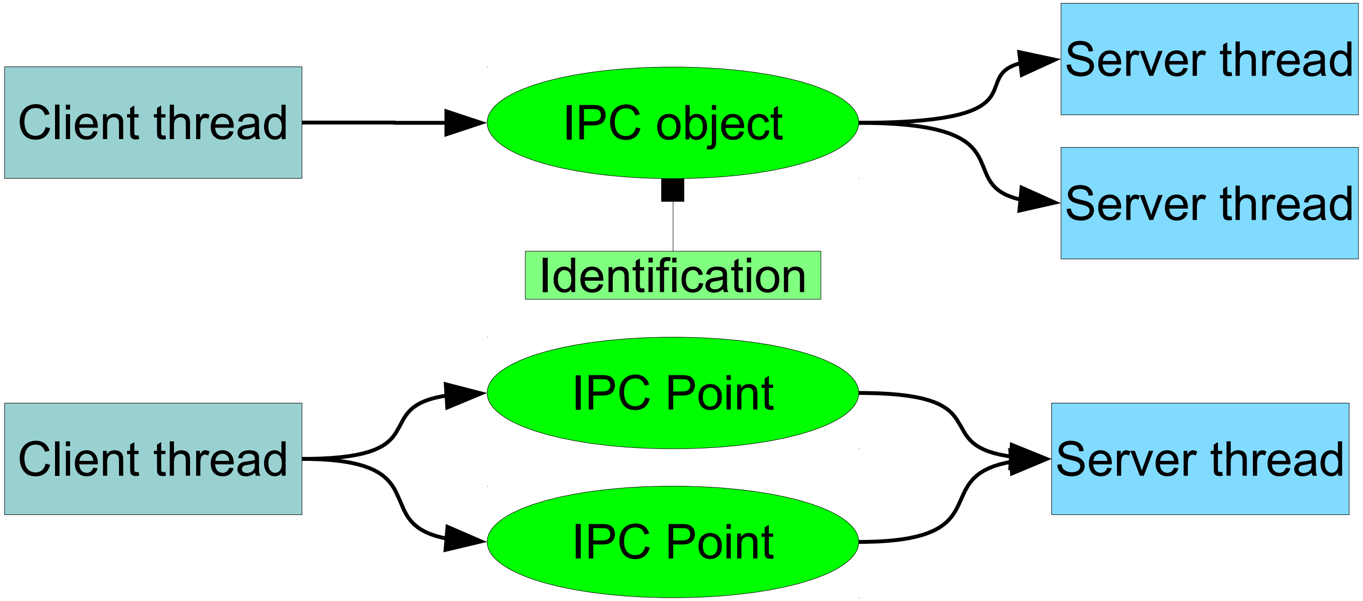
- Connectionless vs. connection-oriented
  - What is the order of delivered messages
- Reliable vs. Unreliable
  - Can a message get lost
- Point-to-point vs broadcast or multicast
  - How many receiver can be addressed
- Asynchronous vs. synchronous
  - Does the sender wait for the receiver
- Buffer vs. unbuffered
  - Message buffering within the kernel
- Direct vs. indirect
  - How is the destination addressed
- Data items
  - What type of data items are transferred

Synchronous vs. Asynchronous IPC

- Synchronous IPC
  - Sender blocks if receiver is not ready
  - Requires no data buffering inside the kernel
  - *Wait queue* for every receiving thread
    - Holds blocked senders for the receiver
    - Requires enqueue/deqeue policy
- Asynchronous IPC
  - Sender does not block if receiver is not ready
  - Requires data buffering inside the kernel
  - Suitable for interrupts because only one bit is transferred

Direct vs. Indirect IPC

- Direct IPC:
  - Sending thread addresses another thread
- Indirect IPC:
  - Sending thread addresses communication object (called: port, portal, gate, endpoint)
  - IPC object redirects message to a receiver thread
  - Advantage 1: Hide the implementation of server threads
  - Advantage 2: Possibility to hold state for client-server connection



IPC Primitives

- Send
  - Send to specific thread/IPC object
- Closed receive
  - Receive from specific thread
- Open receive
  - Receive from any thread
- Send and closed wait
  - Send to and receive from specific thread/IPC object
  - Typical client operation - *call*
- Send and open receive
  - Receive from any thread and send to specific thread
  - Typical server operation – *reply-and-wait*
- Sleep
  - Neither send nor receive
  - Yield until timeout expires

Example: Mutual Exclusion with IPC

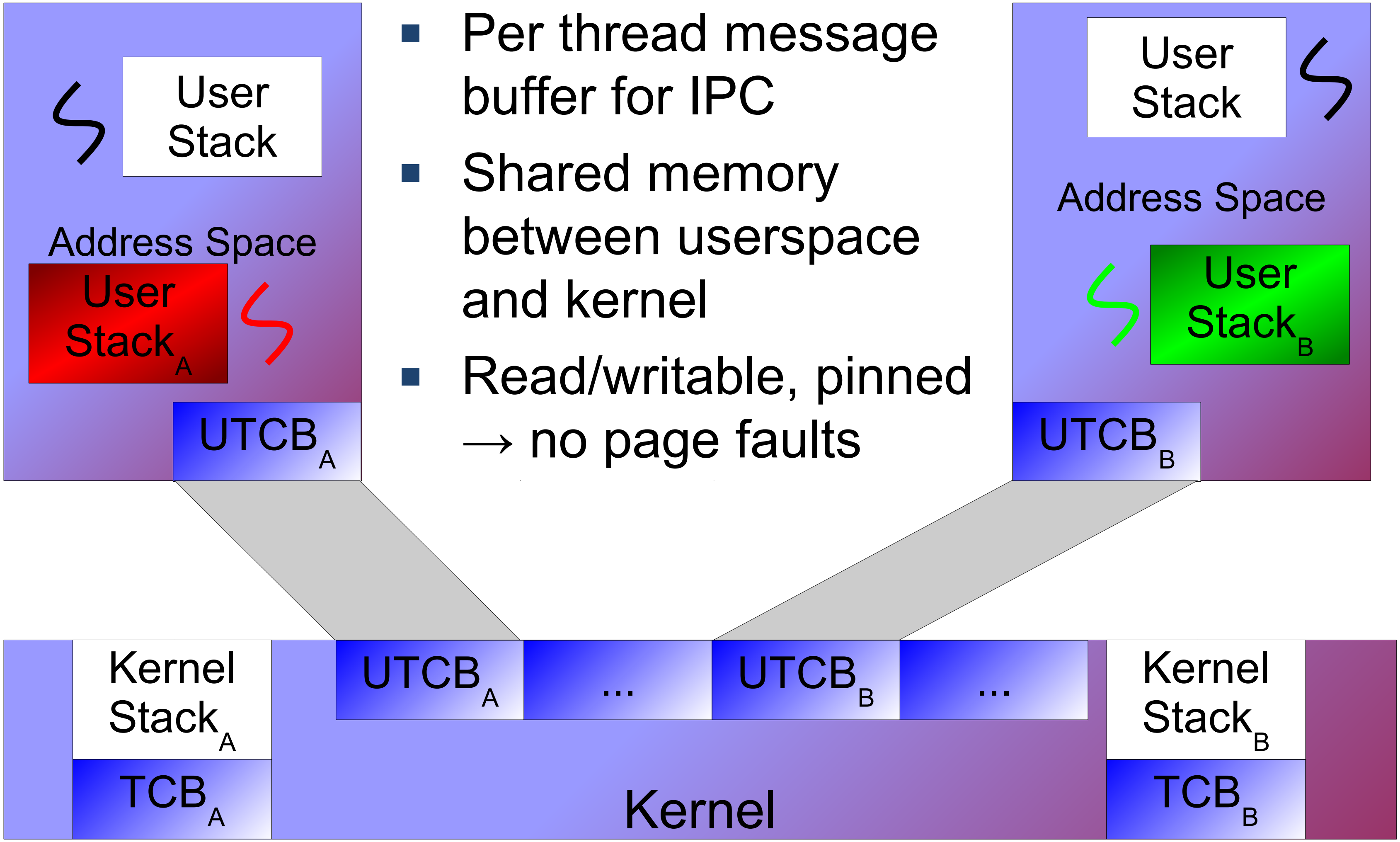
- User-level protocol provides mutual exclusion:
  - *Synchronization thread* protects critical section
  - *Worker thread* calls synchronization thread before entering
  - Synchronization thread replies if critical section can be entered
  - Worker thread blocks if critical section not available
  - Worker thread calls synchronization thread after leaving
  - Synchronization thread sends reply to next waiting worker thread
- Implementation not optimal for performance but correct
- Others:
  - Producer-consumer synchronization together with shared memory
  - Connection-oriented client-server communication

IPC Types

- Register-only IPC
  - Very fast but amount of data is limited to number of registers
- UTCB IPC
  - User Thread Control Block: user accessible, kernel-provided and pinned page, message buffer with guaranteed no page faults
  - Copy data from one UTCB to another UTCB
  - Fast but amount of data is limited to UTCB size
- User-memory IPC
  - Copy of memory areas between user address-spaces
  - Amount of data is not limited
  - Page faults can occur
- Flexpage IPC
  - Mapping of memory areas and capabilities
  - Page fault IPC is special case
- Interrupt IPC
  - Relaying Interrupts and Exceptions as messages

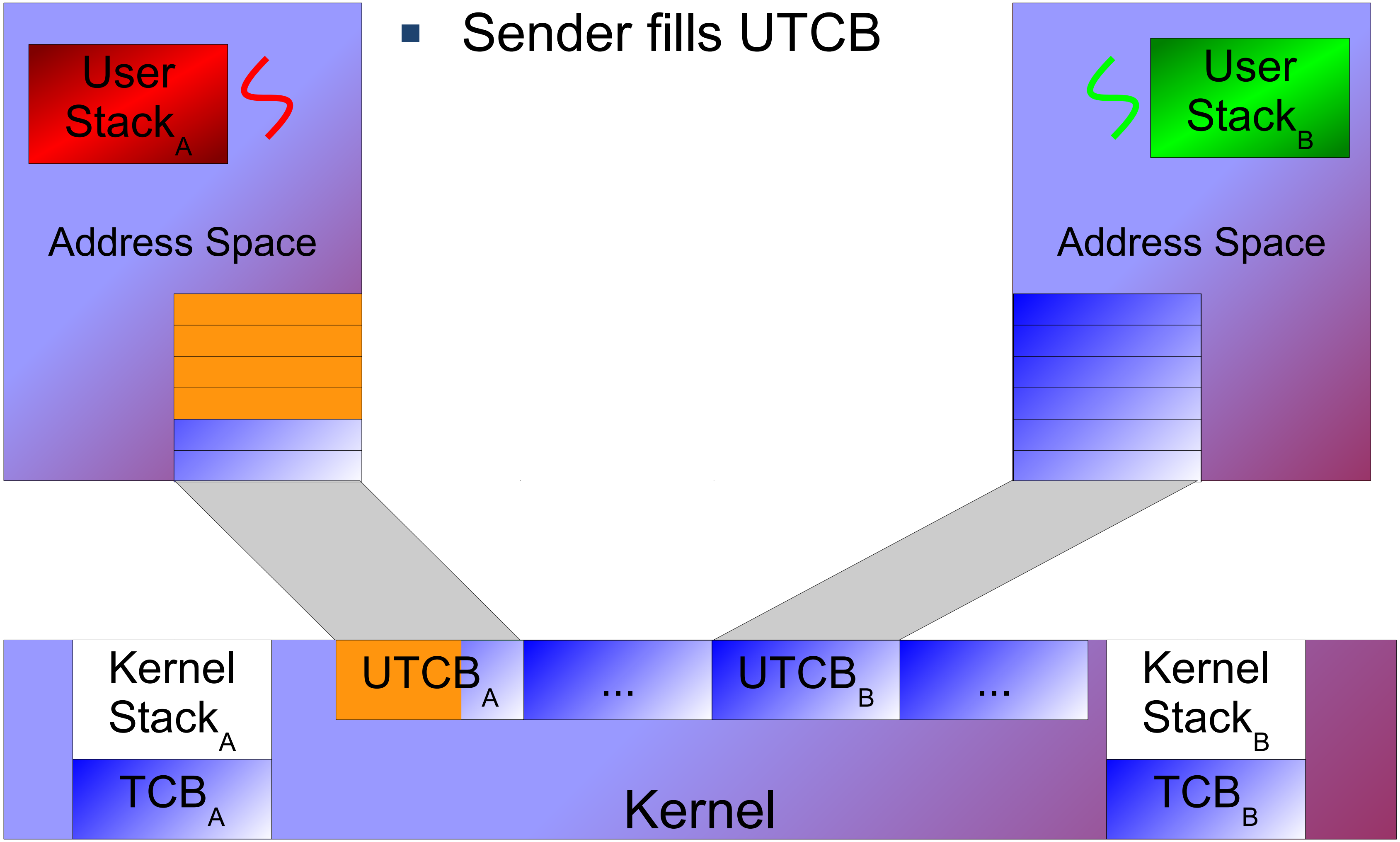
User Thread Control Block

- Per thread message buffer for IPC
- Shared memory between userspace and kernel
- Read/writable, pinned → no page faults



IPC send with UTCB

- Sender fills UTCB

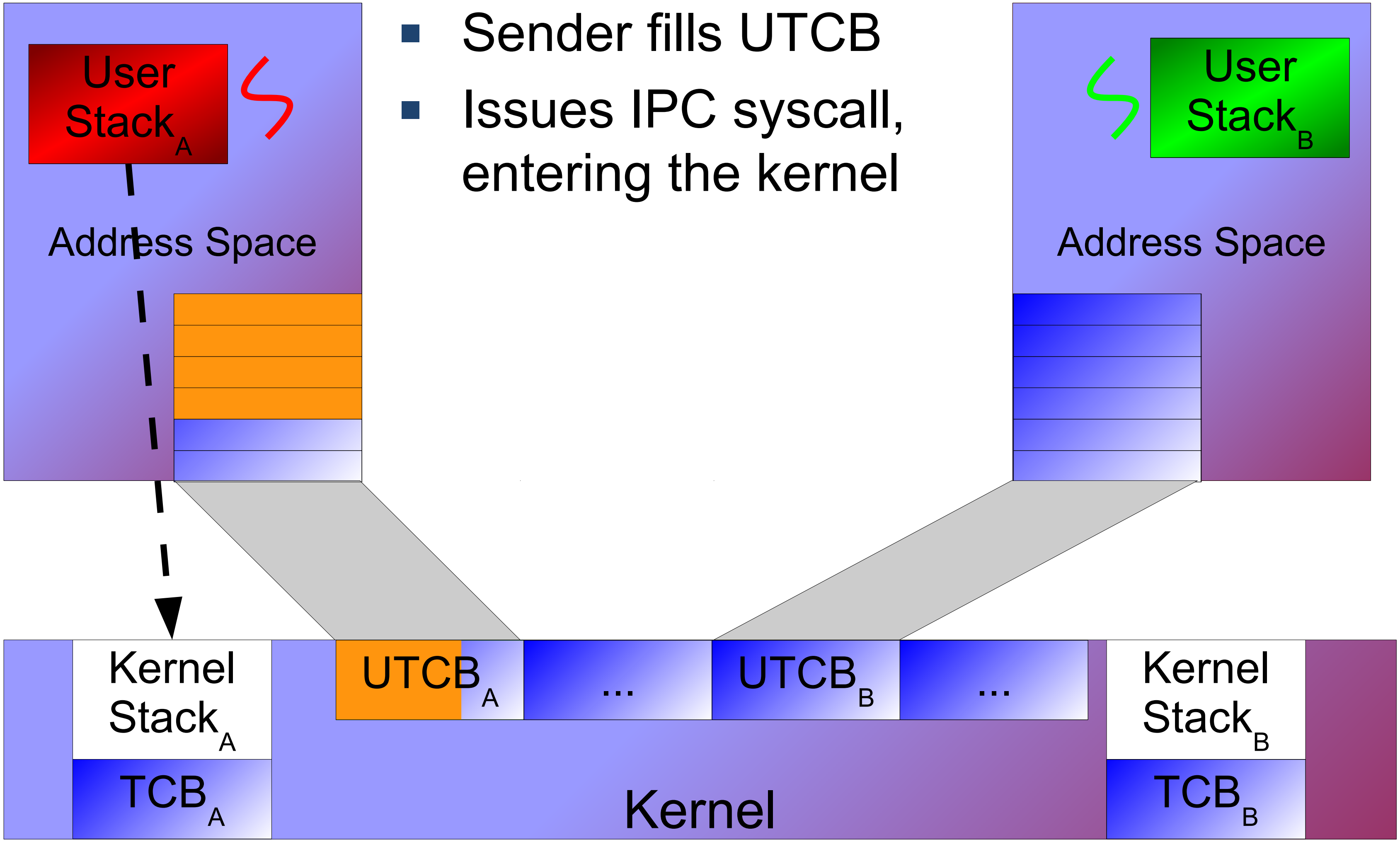


Benjamin Engel
Torsten Frenzel

TU Dresden Operating Systems Group

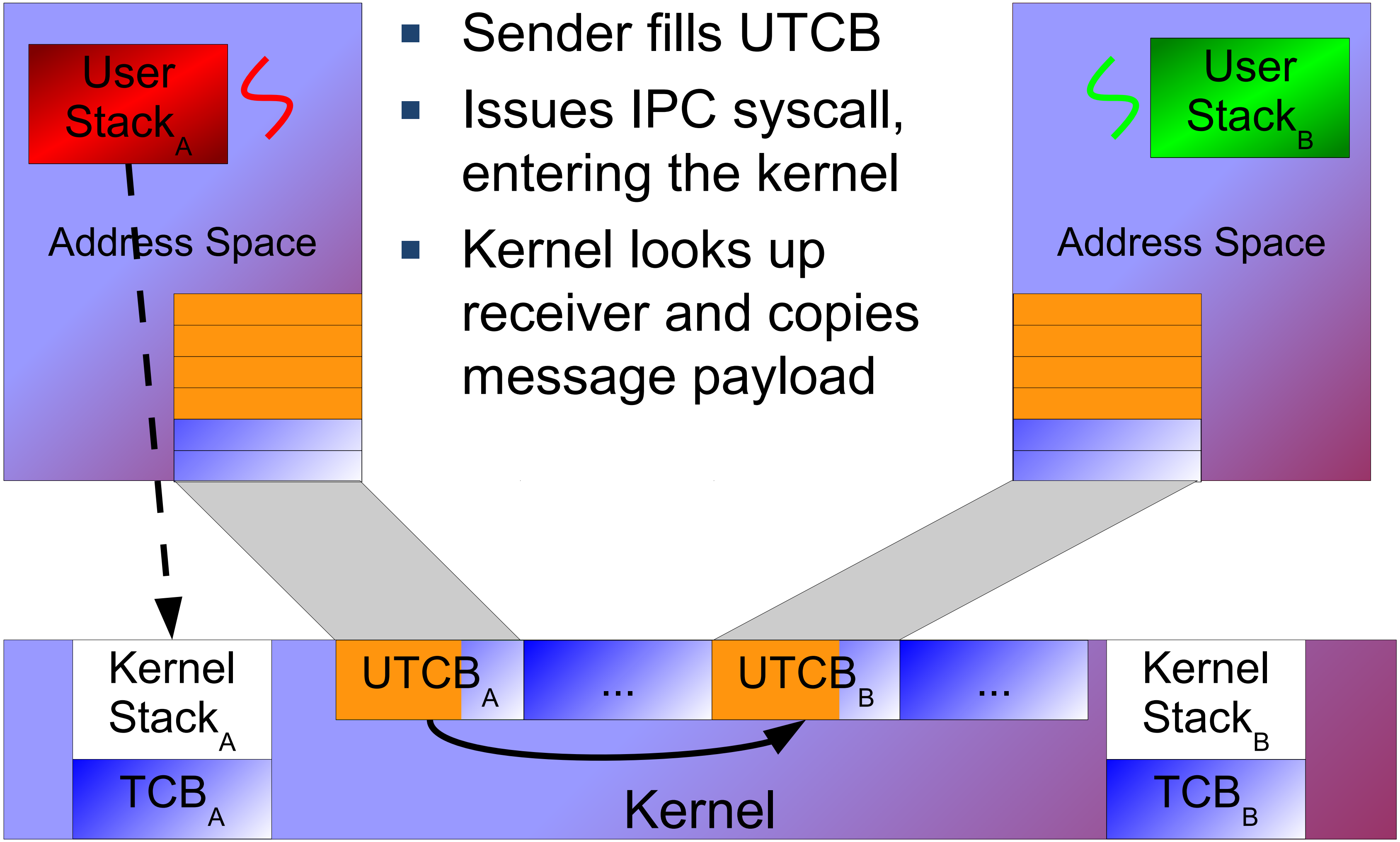
IPC send with UTCB

- Sender fills UTCB
- Issues IPC syscall, entering the kernel



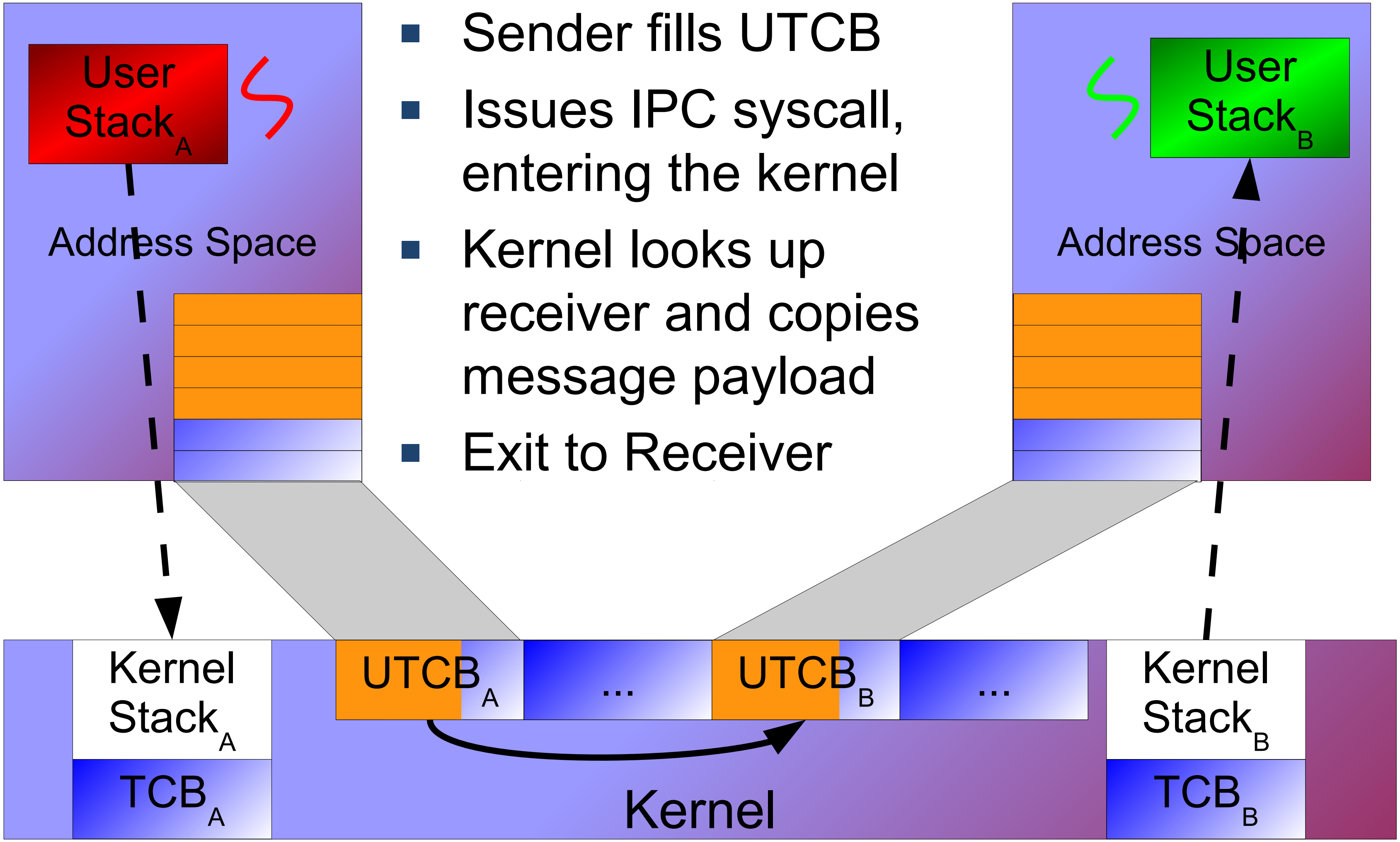
IPC send with UTCB

- Sender fills UTCB
- Issues IPC syscall, entering the kernel
- Kernel looks up receiver and copies message payload



IPC send with UTCB

- Sender fills UTCB
- Issues IPC syscall, entering the kernel
- Kernel looks up receiver and copies message payload
- Exit to Receiver



Benjamin Engel
Torsten Frenzel

Wait, we can do more ...

- Sending pages instead of data → establishing shared memory regions
- Sending resources (precisely: access rights to resources) →granting fine grained IO access to devices
- Sending IPC endpoints → build more communication channels between Thread
- Sending Interrupts → kernel translates interrupts to IPC messags

- Generalizes to capability transfers
  - Cap_Mem → access right to a page
  - Cap_IO → resembles IO port rights
  - Cap_Obj → Kernel objects, like IPC endpoints, Threads, Semaphores, ...

Memory transfer: Flexpages

- Flexpages describe areas in address spaces (size aligned)
- Remember: Segments: Base + Limit
- Flexpage size 2^{size}, smallest is hardware page (e.g. 4KiB)

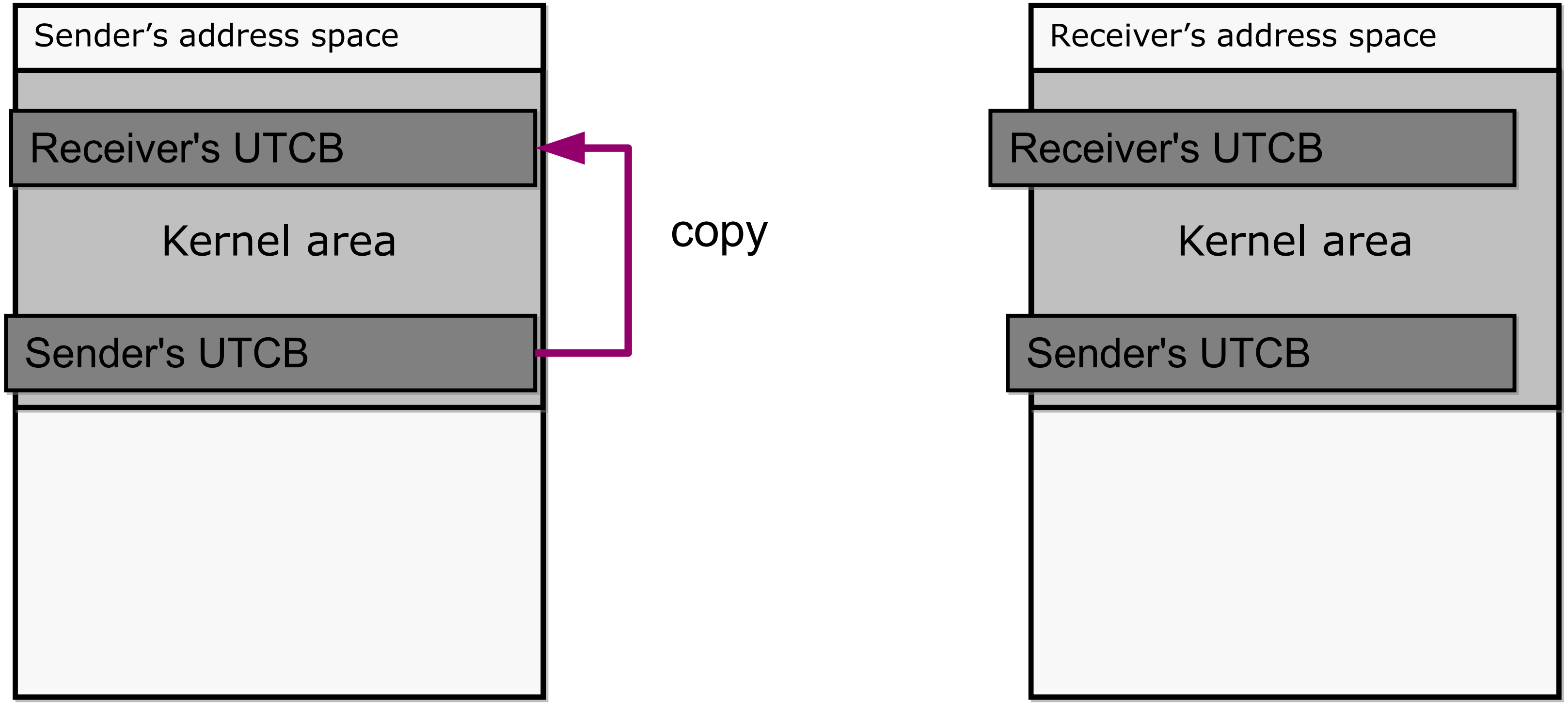
Base	Order	~
31 12	11 7	6 0

- 32bit, 4Kib page size → 20 bit Base
- 5 bit order → 2^0 up to 2^{31}, covers whole address space (6 bit in case of 64 bit address space)
- Sender and Receiver specifies Send/Recv-Flexpage

Base	Order	Size	Address Range
0	0	4KiB	0x00000000 - 0x00000FFF
0x4000	2	16KiB	0x00004000 - 0x00007FFF
0xC0000000	18	1GiB	0xC0000000 - 0xFFFFFFFF
0x4000	3	32KiB	Invalid, not size aligned

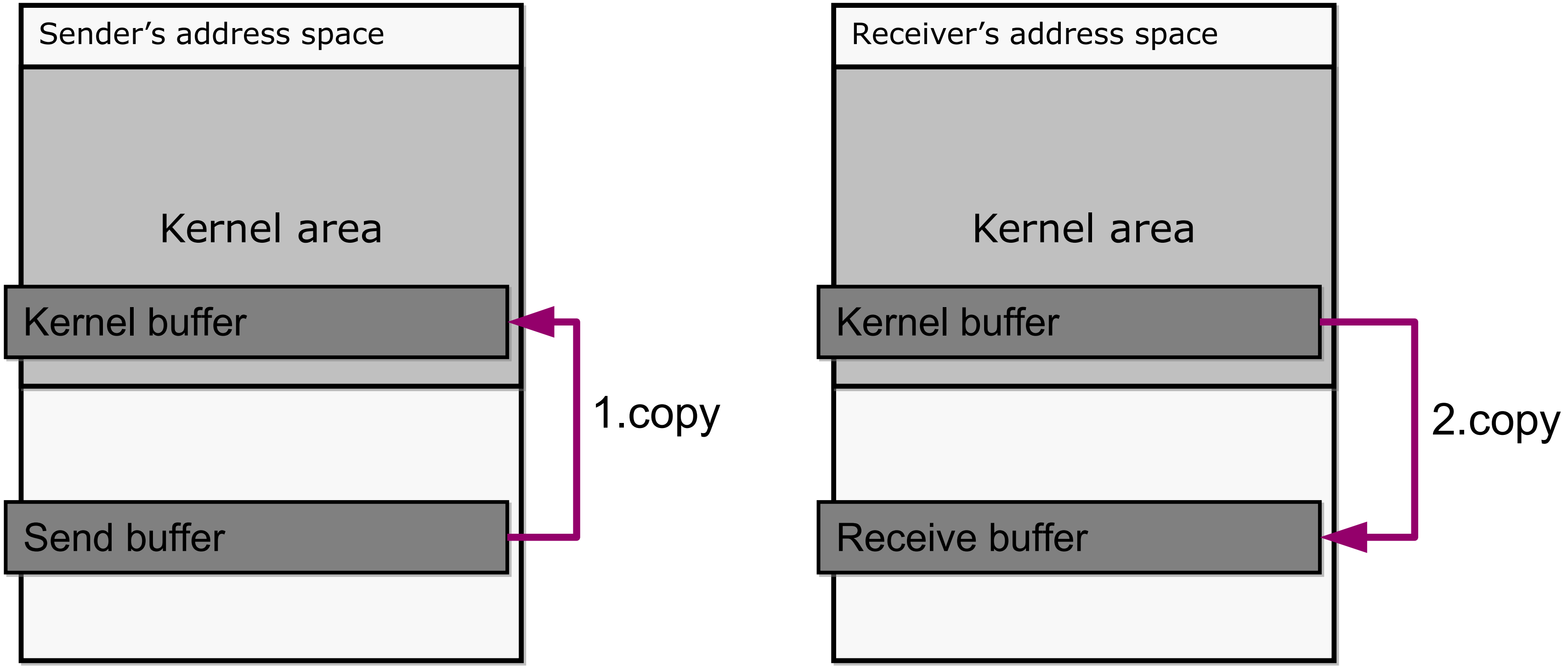
UTCB IPC

- UTCBs hold relevant information of a thread
  - Kernel and user accessible (read and write)
  - Task-local and associated with **one** or more threads
  - No page faults can occur on access
- Can be used as message buffer



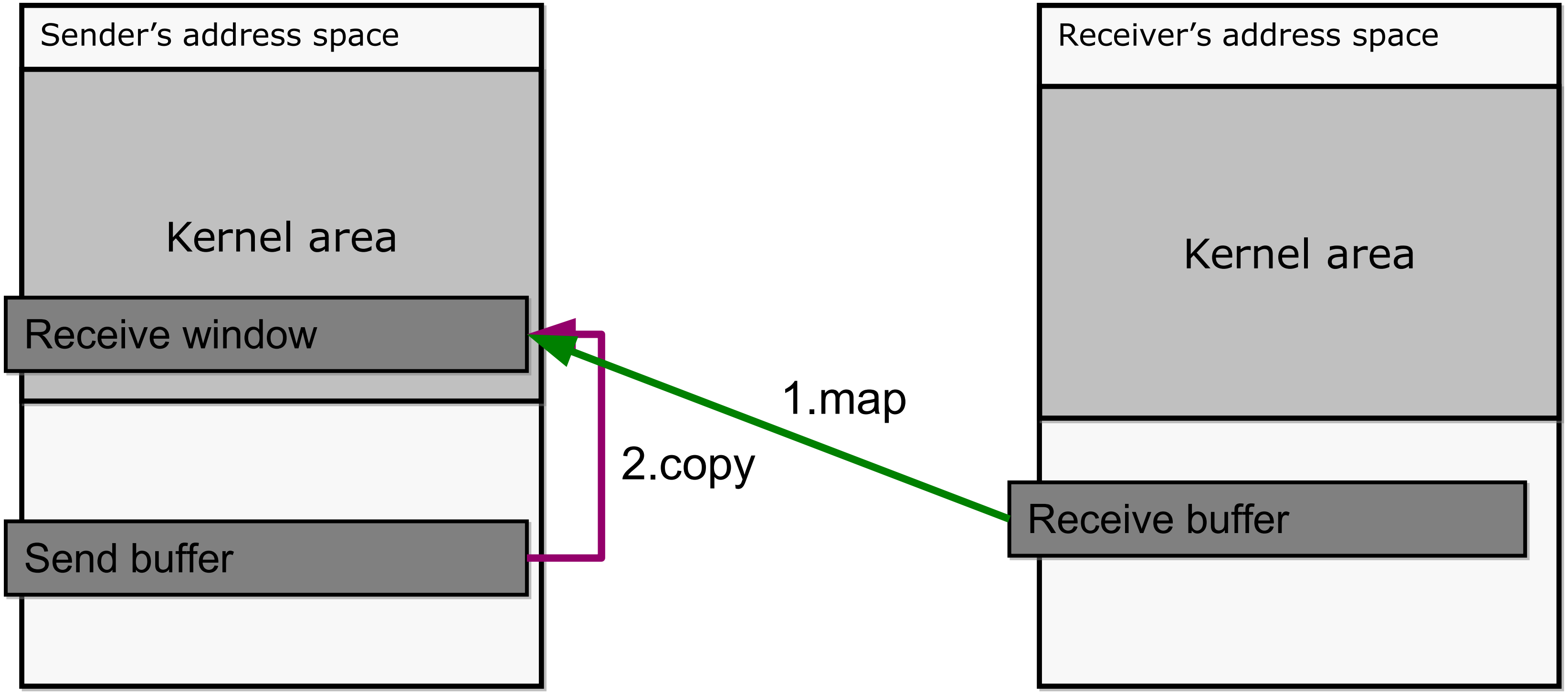
User-Memory IPC with Kernel Buffer

- Two copy operations
  - Copy send buffer to kernel buffer in sender's address space
  - Switch to receiver's address space
  - Copy kernel buffer to receive buffer in receiver's address space



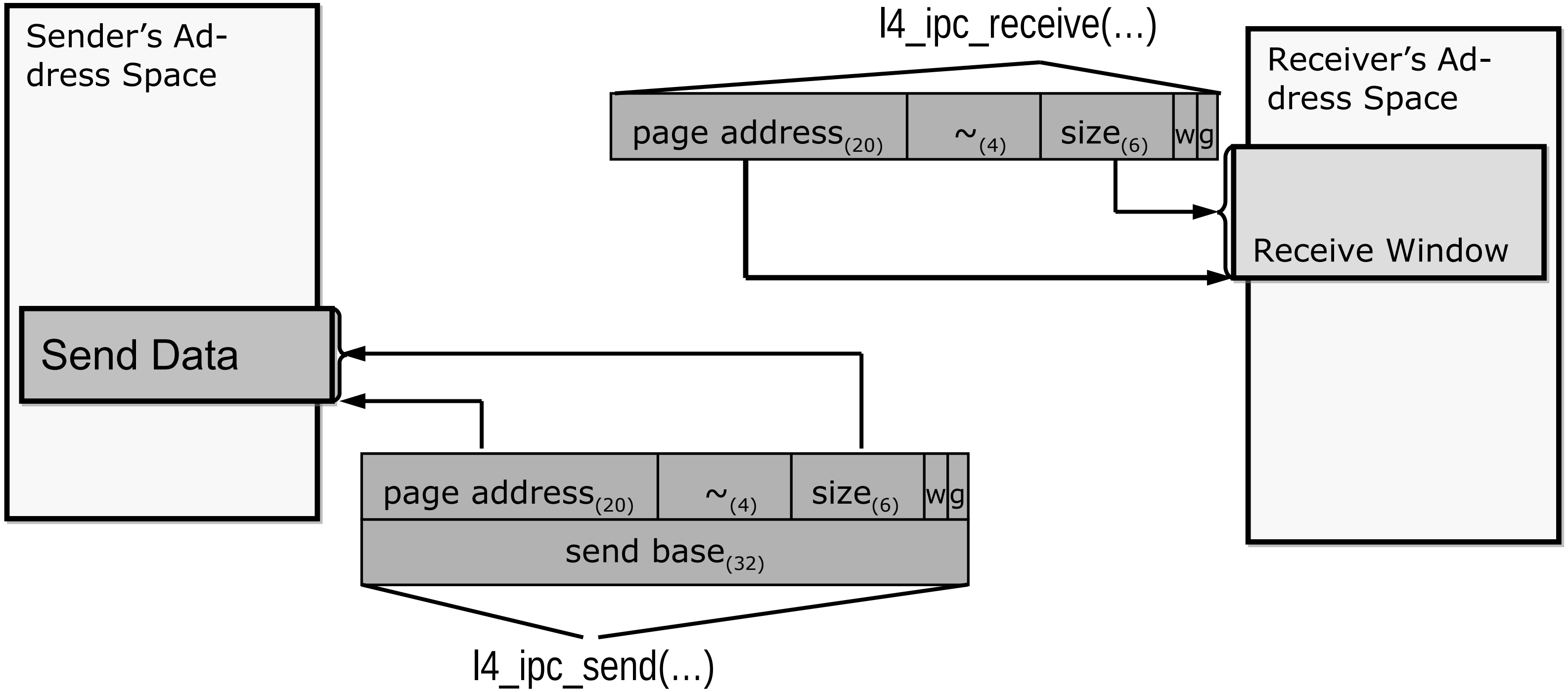
User-Memory IPC with IPC Window

- IPC window provides 'view' into another address space
  - Temporary mapping inside the kernel address space
  - Flushed on each thread switch
- One copy operation
  - Map receive buffer to IPC window
  - Copy send buffer to IPC window



Flexpage IPC

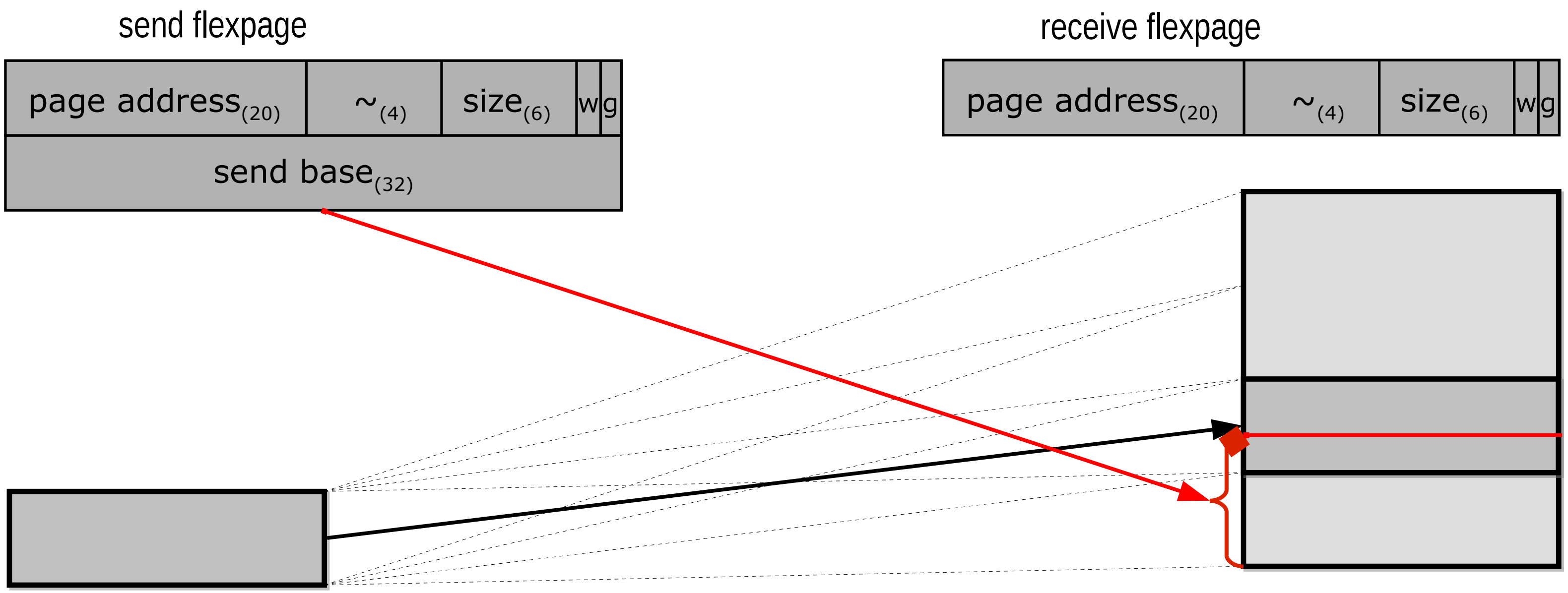
- Sender specifies one or more Send-flexpages
  - Send-flexpage is a flexpage with send base in receive window
- Receiver specifies Receive window flexpage



Flexpage IPC (2)

Send-flexpage is smaller than the receive window

- Target position in receive window is derived from page alignment and send base



send flexpage

page address$_{(20)}$	~$_{(4)}$	size$_{(6)}$	w	g
send base$_{(32)}$				

receive flexpage

page address$_{(20)}$	~$_{(4)}$	size$_{(6)}$	w	g

Example: send flexpage has ¼ size of receive flexpage
→ 4 possibilities to map flexpage into receive window

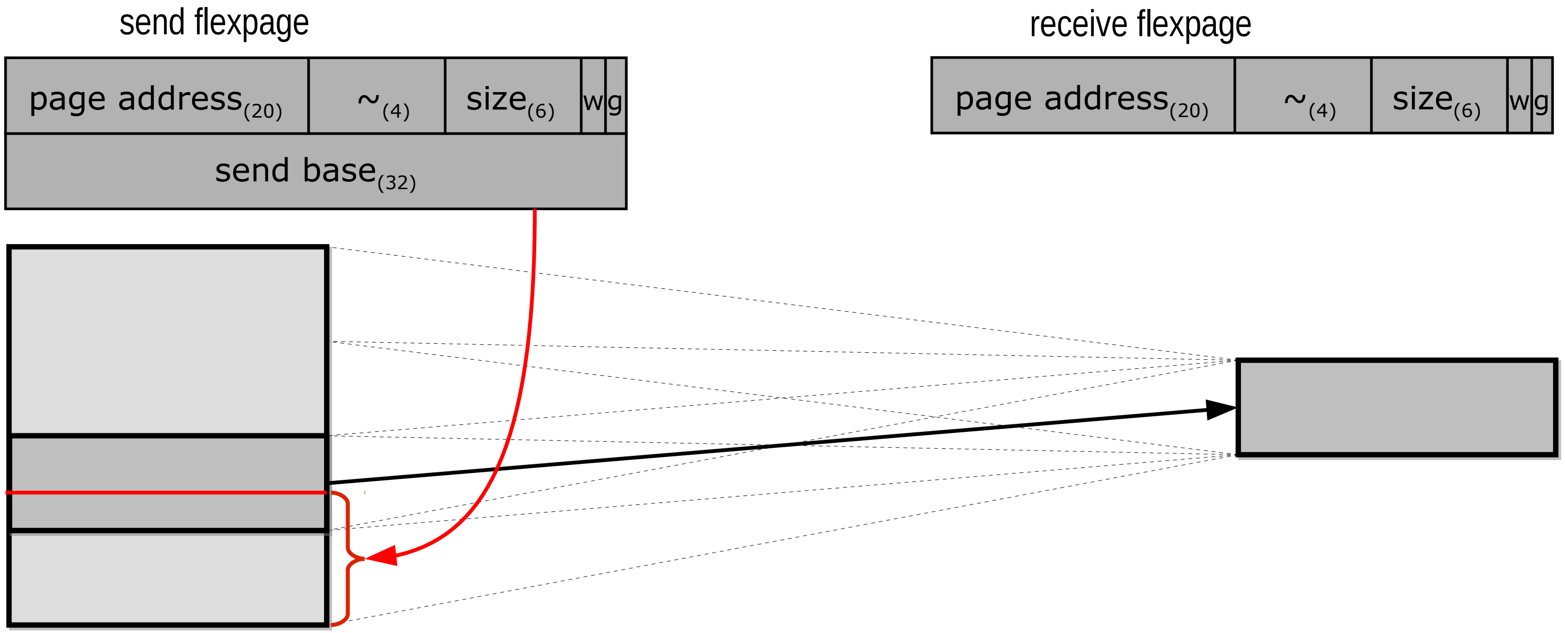
Flexpage IPC (3)

Send flexpage is larger than receive window

- Source position in send flexpage is derived from page alignment and send base

⇨ *send base* is a hot spot for mapping IPC

⇨ Actual *send base* depends on information about the receiver



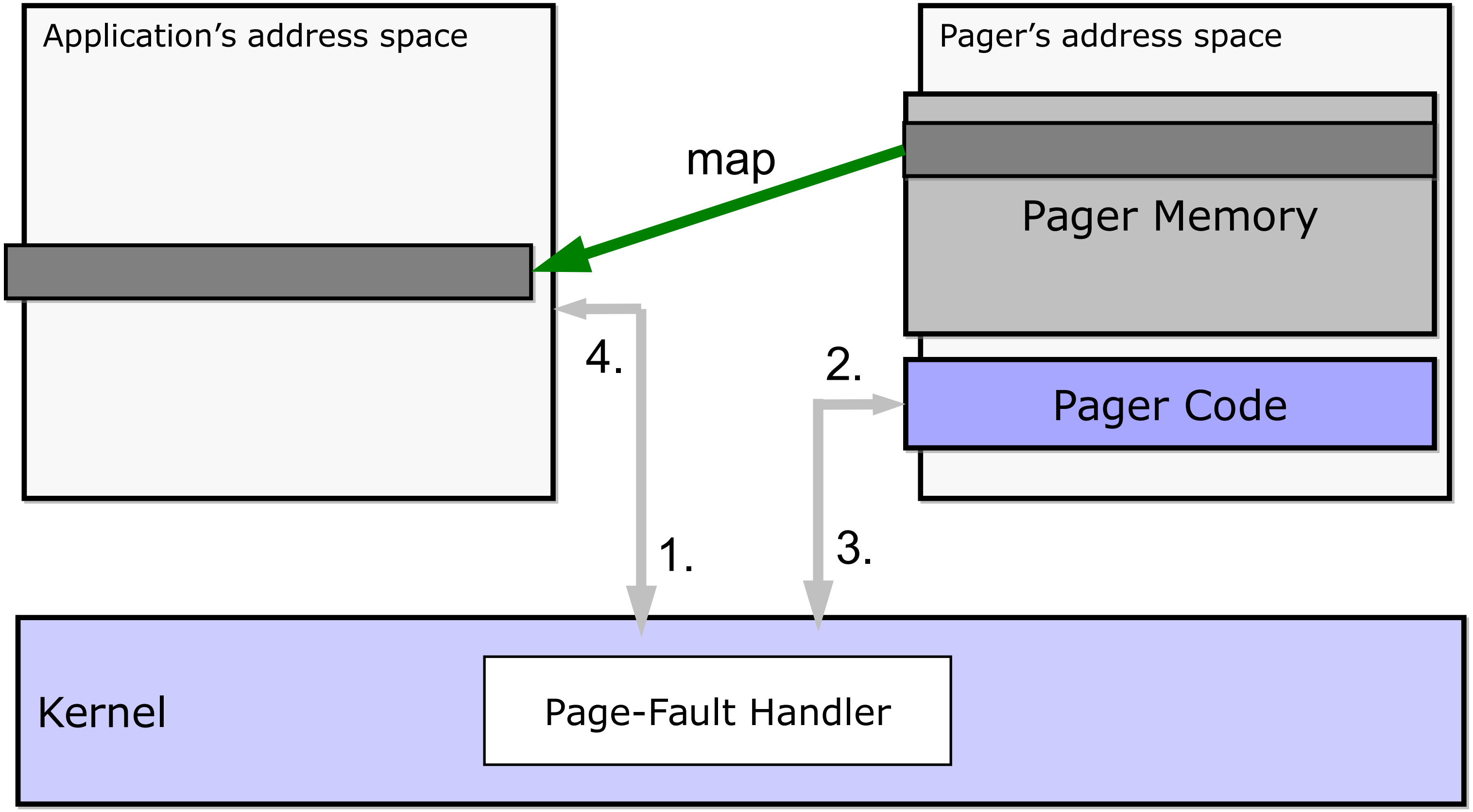
Example: receive flexpage has ¼ size of send flexpage

Exception IPC

- For every exception to be handled, have a thread ready to receive, e.g. x86 exception #14 → page fault handler
- Kernel prepares IPC message with selected architectural state, in case of page faults: faulting address
- Kernel opens whole faulting address space to receive a flexpage, thus accepting any mapping
- Page fault handler thread replies with flexpage to resolve page fault

Pagefault Handling

- General pagefault mechanism:
  1. Application causes a page fault
  2. Kernel sends Pagefault IPC to pager thread
  3. Pager thread replies with flexpage IPC
  4. Application receives mapping and resumes

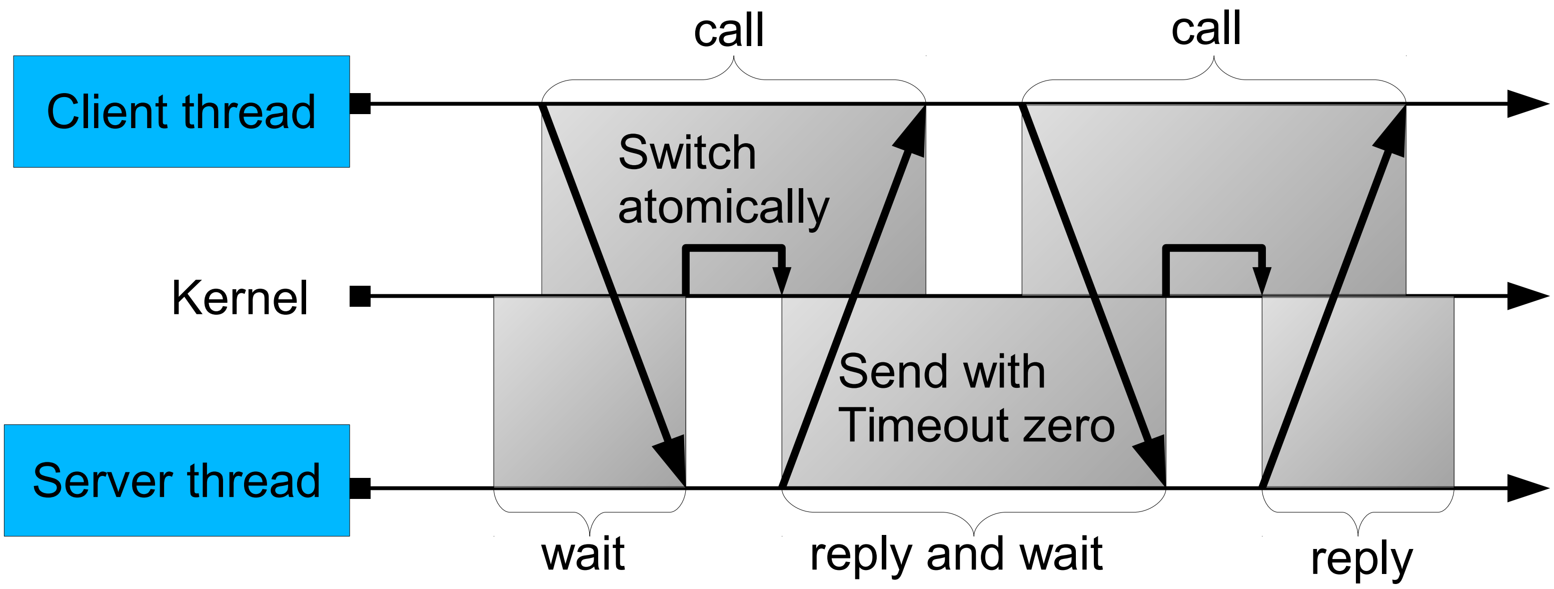


Interrupt IPC

- Kernel has driver to program interrupt controller
- Interrupts from peripheral devices (disk,network, …) preempt current activity and trap in the kernel
- Interrupts are implemented as special sender object
  - Threads can *attach* itself to interrupts
  - Thread can wait and receive message from an interrupt
  - Thread can wait on many interrupts using open wait
  - One bit of information is transferred
  - Interrupt is enqueued into sender queue of receiver thread if thread is not ready
- Interrupt source is disabled after it is has triggered and needs to be explicitly enabled by the receiver

Switching from Send to Receive

- Switch from send part to receive part **atomically**
- Why prepare receive part atomically?
  - – Servers do not trust clients
  - → Servers reply with timeout zero
  - → Client needs to be ready to receive immediately after sending
  - → Flip one bit to switch from send to receive part



Pagefaults during IPC

- Long IPC can touch user memory
  - ➔ Page faults can occur if virtual address is not mapped to physical memory
  - – Page fault in sender's address space
  - – Page fault in IPC window (receiver's address space)
- Page fault handler suspends ongoing IPC operation
  - – Save current IPC state (somewhere)
  - – Setup *nested* page fault IPC
    - ➔ Start another IPC during ongoing IPC
  - – If page fault IPC succeeded:
    - Restore suspended IPC operation
  - – If page fault IPC failed:
    - Abort ongoing IPC operation
- Even more complicated if ongoing page fault IPC is canceled

IPC Termination

- Normally
- Abnormally because of non-existing partner
- Abnormally because of memory copy failure
  - Due to too small send or receive buffer
- Abnormally because of memory map failure
  - Due to shortage of page tables
- IPC partner is canceled by another thread
  - Cancel operation executed before rendezvous
- IPC partner is aborted by another thread
  - Cancel operation executed after rendezvous
- IPC partner is killed by another thread
- Timeout occurs
  - Send/receive IPC timeout
  - Send/receive page fault timeout